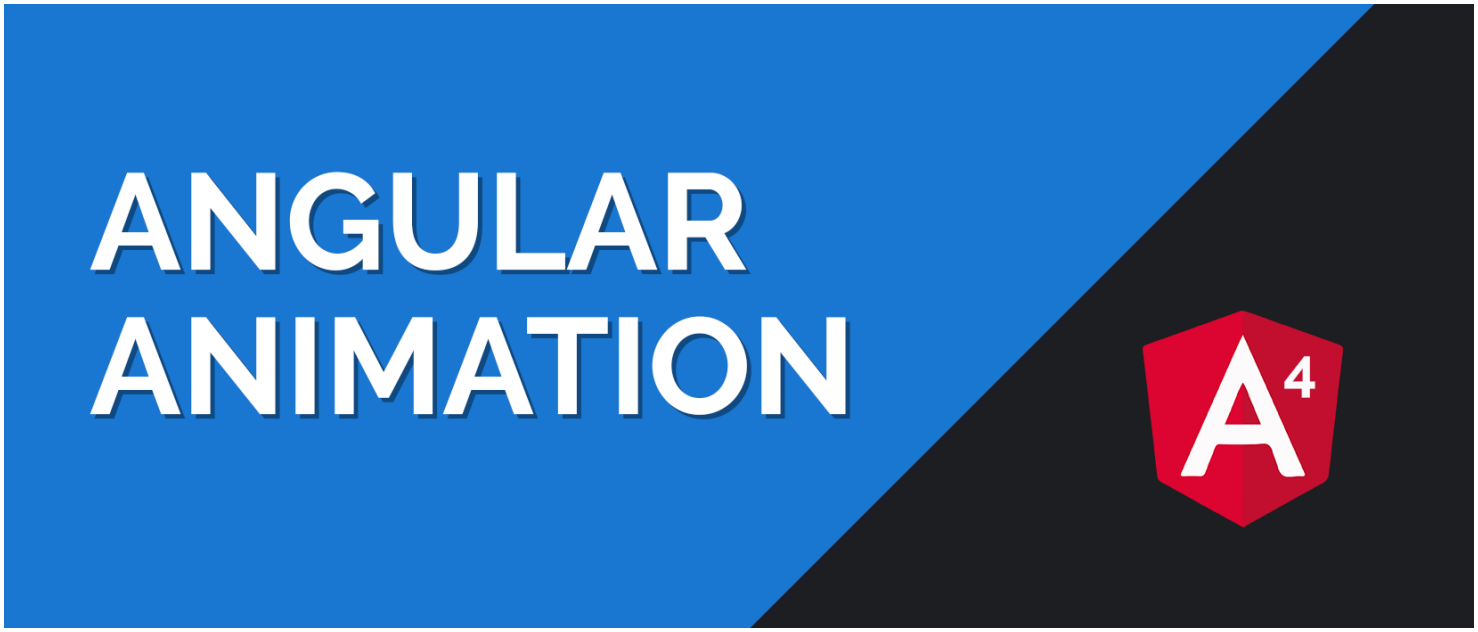
Angular 4 Animation Tutorial



Angular offers the ability to create powerful animations and activate them based on a variety of factors. You can place animations on any HTML elements, and make them occur based on Angular lifecycles, user events and more.

One of the changes from Angular 2 to 4 was the fact that they moved the animation functions from the angular core library, into their own animations library. This means that if you wish to integrate animations within your Angular 4 app, you will need to use npm to install the animations package, as well as import the library within the app.module.ts file.

So, let's get started with that, and then we'll figure out how Angular 4 animations work.

Installing Angular Animations

Being that this tutorial is a part of our Free Angular 4 Course, we already have a project setup and ready to go. If you've landed on this tutorial page without having followed along with the previous videos, make sure you use the Angular CLI to generate a new Angular project. This way, we'll all be on the same page.

In the project folder of your Angular app, at the console, type:

```
> npm install @angular/animations@latest --
save
```

This will save the latest version of the angular animations library, as well save it as a dependency in the package.json file.

Next, in the **/src/app/app.module.ts** file, we add the import:

```
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';
```

And in the imports array of the @NgModule decorator, we add:

```
@NgModule({

  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,

BrowserAnimationsModule
  ],
})
```

Great. Now we can proceed as we normally would as if this were an Angular 2 app.

Getting your component ready for animation

First, we need to import some animation functions to the top of the intended component:

```
import { trigger,state,style,transition,animate,keyframes } from
'@angular/animations';
```

For animation to work, all of these functions with exception to **keyframes** is required. If your component template won't include multi-step animations, then you can omit keyframes.

Next, we have to add the **animations** property to the **@Component()** decorator:

```
@Component({
  selector: 'app-
root',
  template: `
  <p>I will
animate</p>
  `,
  styles: [``],
  animations: [



  ]
})
```

Just to make things simple, we're using inline HTML and CSS here.

Next, let's define an actual animation.

Animation Triggers

Remember when we imported a **trigger** function up top? The **trigger** animation function is the starting point of each unique Animation animation.

The first argument accepts the name of the animation, and the second argument accepts all of the other animation functions that we imported.

So, within the **animations** property of the component decorator, we add:

```
animations: [
    trigger('myAwesomeAnimation',
[

    ]),
  ]
```

Notice the comma after the trigger function? That's there so that you can define multiple animations, each with unique animation names defined within the first argument.

Next up are animation states.

Animation States & Styles

The state function allows you to define different states that you can call and transition between. The first argument accepts a unique name, and the second argument accepts the style function.

The style function allows you to apply an object with web animation property names and associated values.

So, expanding on our previous example:

```
animations: [
    trigger('myAwesomeAnimation', [
        state('small', style({
            transform: 'scale(1)',
        })),
        state('large', style({
            transform:
'scale(1.2)',
        })),
    ]),
  ]
```

We have 2 different states where the **scale** property is going from a default size of 1, to 1.2. Next up is the **transition** function.

Animation Transitions

The transition function is what makes the actual animation occur. The first argument accepts the direction between 2 different states, and the second argument accepts the **animate** function.

The animate function allows you to define the length, delay, and easing of a transition. It also allows you to designate the **style** function for defining styles while the transitions are taking place, or the **keyframes** function for multi-step animations; both of which are placed in the second argument of the animate function.

For now, we will omit the second argument of the animate function.

```
animations: [
    trigger('myAwesomeAnimation', [
        state('small', style({
            transform: 'scale(1)',
        })),
        state('large', style({
            transform: 'scale(1.2)',
        })),
        transition('small => large', animate('100ms ease-
in')),
    ]),
  ]
```

This transition is defining the length of the transition, as well as the easing type when an HTML element that uses the **myAwesomeAnimation** trigger goes from the **small** state to the **large** state.

Let's leave it like this for now, but we will revisit this code snippet shortly.

Attaching the Animation in the Template

Our template is very simple. We just have a single paragraph element. Let's attach the animation to that element and add a click event bound to a method.

```
template: `
  <p [@myAwesomeAnimation]='state' (click)="animateMe()">I will
animate</p>
  `,
```

As you can see, to attach an animation to an HTML element, you wrap brackets around the trigger name, preceded by an @. You bind it to a template expression, which in this case is a property that we will define in the component class.

Before we do that, let's give our paragraph some style in the styles property:

```
styles: [`
  p {
    width:200px;

background:lightgray;
    margin: 100px auto;
    text-align:center;
    padding:20px;
    font-size:1.5em;
  }
  `],
```

Next, let's work on the component.

Making it Work in the Component

Now we have to define the state property and the method. So, in the component class, let's add:

```
export class AppComponent {
  state: string = 'small';

  animateMe() {
        this.state = (this.state === 'small' ? 'large' :
'small');
  }

}
```

When the app loads, we bind the **state** property to the small state of our animation.

Then when a user clicks on the paragraph element, it will call the **animateMe()** method. Inside, it simply toggles back and forth between small and large.

Save it, run **ng serve** in the console in the project folder, and watch it work!

The animation at this point only works in one direction, because we have only set 1 transition property going from **small** to **large**.

To make it work in both directions, you change **small => large** to **small <=> large**, which means in both directions.

Save it and now both state changes are animating.

Adding Style During Transitions

As mentioned earlier, you can add a second property on to the **animate** function which accepts the **style** function. Let's do that.

```
        transition('small <=> large', animate('300ms ease-in',
style({
          transform: 'translateY(40px)'
        }))),
```

If you save, you will see during the transition, the paragraph element moves down 40 pixels.

Adding Keyframe Animations

The second argument of the animate function also accepts the **keyframes** function. This allows you to create elaborate, sequence-based animations.

```
        transition('small <=> large', animate('300ms ease-in',
keyframes([
          style({opacity: 0, transform: 'translateY(-75%)', offset: 0}),
          style({opacity: 1, transform: 'translateY(35px)',  offset:
0.5}),
          style({opacity: 1, transform: 'translateY(0)',     offset:
1.0})
        ]))),
```

The keyframes function accepts an array of style functions, which includes animation properties and an offset. The

offset designates the point at which the next style function begins during the animation. 0 is the beginning and 1 is the end.

This, in a nutshell, is how animations work in Angular.